VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DYNAMICKÁ REKONFIGURACE HARDWAROVÝCH AKCELERÁTORŮ

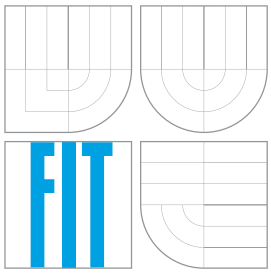
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ BRABEC

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DYNAMICKÁ REKONFIGURACE HARDWAROVÝCH AKCELERÁTORŮ

DYNAMIC RECONFIGURATION OF HARDWARE ACCELERATORS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ BRABEC

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. KAREL MASAŘÍK, Ph.D.

BRNO 2013

Abstrakt

Práce se věnuje využití dynamické rekonfigurace FPGA v oblasti aplikačně specifických procesorů, a to zejména vzhledem k rychlosti jejich vývoje, možnostem akcelerace výpočtů a univerzality. Dále je navrženo rozšíření aplikačně specifického procesoru Codix o rekonfigurovatelnou jednotku a popsána její implementace. V závěru jsou shrnuty získané poznatky a nastíněny možnosti dalšího vývoje.

Abstract

Thesis deals with usage of dynamic reconfiguration of FPGA in area of application specific instruction-set processors, considerng time-to-market, possibilities of acceleration and universality. Furthermore, it is designed an extension of application specific processor Codix with reconfigurable unit and it is described its implementation. Finally, the results are evaluated and opportunities for further development are identified.

Klíčová slova

Dynamická rekonfigurace, hardware/software co-design, ASIP, FPGA, CodAL, Codix

Keywords

Dynamic reconfiguration, hardware/software co-design, ASIP, FPGA, CodAL, Codix

Citace

Lukáš Brabec: Dynamická rekonfigurace hardwarových akcelerátorů, bakalářská práce, Brno, FIT VUT v Brně, 2013

Dynamická rekonfigurace hardwarových akcelerátorů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Karla Masaříka, Ph.D.

.....
Lukáš Brabec
14. května 2013

Poděkování

Děkuji panu Ing. Adamu Husárovi za odborné konzultace, pomoc při řešení návrhu a další poskytnuté rady. Dále děkuji Ing. Karlu Masaříkovi, Ph.D. za vedení práce.

Obsah

Kapitola 1

Úvod

Výpočetní technika se čím dál více stává každodenní součástí života. Velký nárůst v nedávné době zaznamenaly zejména chytré telefony, tablety a jiná vestavěná zařízení, a to v takové míře, že se začíná hovořit o tzv. „post PC éře“ [6]. Dle prognózy bude do roku 2017 prodej tabletů vyšší, než prodej tradičních počítačů [10]. Ruku v ruce s tímto trendem jde i zvýšená potřeba analýzy a návrhu integrovaných obvodů pro specifické oblasti využití vestavěných zařízení.

Pro specifické potřeby, jako je např. audio, video a šifrování, jsou v domácích počítačích obvykle použity rozšiřující periferie ve formě karet. Ve vestavěných systémech je jednou možností využít aplikačně specifické obvody (ASIC – application-specific integrated circuit). Tyto obvody jsou navržené přímo na míru dle potřeb dané oblasti a mají vysoký výpočetní výkon vzhledem ke spotřebě. Nevýhodou je velká počáteční investice do výrobní linky spojená s malou univerzálností. Druhou možností je využití procesorů s aplikačně specifickým rozšířením instrukční sady (ASIP – application-specific instruction-set processor), které jsou kompromisem mezi univerzálním procesorem a ASIC.

Ve vestavěných systémech se využívá integrace mikroprocesoru, periferií a paměti na jeden čip - System On the Chip (SoC). Důležitým faktorem SoC je rychlost uvedení na trh. Rychlejší uvedení bude snižovat výslednou cenu čipu a také jeho použitelnost vzhledem k aktuálním požadavkům na trhu [9]. Dobu uvedení na trh lze snížit využitím programovatelných hradlových polí (FPGA - field programable gate array), u kterých je ve srovnání s ASIC a ASIP vyzdvihována právě tato vlastnost [3]. Možností je i využití FPGA v SoC a zejména dynamické rekonfigurace FPGA. Tento přístup umožňuje snížit plochu čipu a zvýšit univerzálnost při zachování aplikační specificity.

Tato práce se zabývá využitím dynamické rekonfigurace FPGA v oblasti aplikačně specifických procesorů, a to zejména vzhledem k rychlosti vývoje ASIPů, možnostem akcelerace výpočtů a univerzality.

Ve druhé kapitoleje představen souběžný návrh hardwaru a softwaru (hardware/software co-design), motivace k tomuto přístupu, jazyk CodAL a mikroprocesor Codix.

Třetí kapitola popisuje FPGA a možnosti a přístupy spojené s částečnou dynamickou rekonfigurací.

Ve čtvrté kapitole je navržena struktura, funkce a připojení rekonfigurovatelné funkční jednotky, která je rozšířením aplikačně specifického procesoru Codix.

Pátá kapitola popisuje implementaci rekonfigurovatelné jednotky v jazyce VHDL a popis práce se softwarem Xilinx zajišťujícím syntézu, implementaci a programování.

Šestá kapitola popisuje použití rekonfigurovatelné jednotky a její otestování, dále shrnuje výsledky rekonfigurace za běhu programu.

Předposlední, sedmá kapitola, diskutuje výsledky z použití rekonfigurovatelné jednotky, jejího provozu, změny funkce za chodu a zejména možnosti dalšího rozvoje do budoucna.

V závěru práce jsou shrnuty poznatky popsané v předchozích kapitolách.

Kapitola 2

Hardware/software codesign

Při návrhu vestavěných zařízení, zejména obsažených SoC, se zohledňují různá omezení jako je velikost, spotřeba elektrické energie, spolehlivost a cena. Dále musí vývoj projít přes fáze verifikace, výroby, ukládání do pouzder a testování [3]. Pro tvorbu softwarového vybavení je třeba mít čip k dispozici (nebo alespoň emulátor). Hardware a software je tak vyvíjen odděleně.Hardware je tvořen jako první a v případě výskytu chyb se musí čip znova vyrobit. Nutnost vyrobit čip opětovně se objevuje i v případě změn, je proto kladen velký důraz na co největší dodržování již daného návrhu [1].

Výše uvedený způsob vývoje hardwaru a patřičného softwaru je v rozporu s potřebou rychle reagovat na požadavky trhu. Aby se dal vývojový cyklus zkrátit, je třeba k vývoji přistupovat jiným způsobem. Od devadesátých let minulého století se začíná využívat přístup souběžného návrhu hardwaru a softwaru, tzv. hardware/software codesign [11].

2.1 Jazyky pro popis architektury

Pro potřeby hardware/software codesignu se používají jazyky pro popis architektury (ADL – architecture description language), které na rozdíl od jazyků popisujících hardware (HDL – hardware description language) obsahují další dodatečné informace týkající se softwarových nástrojů [9]. Právě tyto dodatečné informace umožňují z popisu architektury vytvořit překladač, debugger, simulátor a další.

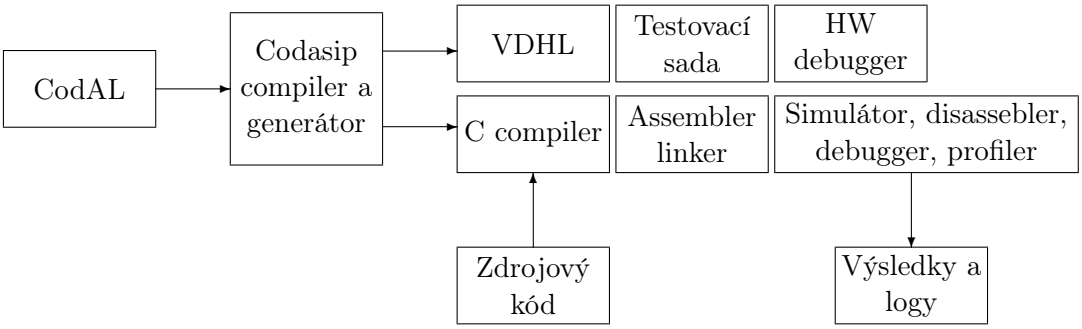
Existují tři kategorie ADL jazyků dělicí na [5, 9]:

- popisující instrukční sadu,
- popisující strukturu,
- smíšené.

Jazyky popisující instrukční sadu jsou často jen výčtem instrukcí a popisu jejich chování. Slouží především ke generování překladačů. Tvorba HDL popisu mikroarchitektury je bez dalších informací nemožná.

Strukturální jazyky obsahují informace potřebné pro generování hardwaru. Bez popisu chování je ale ztížena tvorba překladačů vyšších jazyků.

Smíšené jazyky obsahují informace jak o chování, tak i o struktuře. Lze je tedy použít ke generování výsledného hardwaru a stejně tak ke tvorbě překladačů. Jazyk CodAL, kterým se tato práce mimo jiné zabývá, patří mezi smíšené ADL jazyky.



Obrázek 2.1: Schéma generování nástrojů a HW reprezentace z popisu architektury

2.2 CodAL

Jazyk CodAL byl vyvinut k rychlému prototypování aplikačně specifických procesorů a je navržen pro potřeby souběžného návrhu hardwaru a softwaru. Na základě popisu architektury v jazyce CodAL je vygenerován vnitřní XML kód. Z něj jsou vygenerovány nástroje pro programování a simulaci (obrázek 2.1).

Dále je možné vygenerovat popis hardware v jazyce VHDL použitelný k implementaci na FPGA. Opakované rychlé generování překladačů, simulátorů a popisu hardware umožňuje návrháři rychle procházet návrhový prostor (DSE – design space exploration) a upravovat tak mikroarchitekturu, měnit komunikaci mezi jádry a celkově optimalizovat podle cílového použití. Informace zaznamenané v jazyce CodAL lze rozdělit do čtyřech kategorií [2]:

- popis instrukční sady ve formě gramatiky,
- popis časování mikroarchitektury,
- popis chování jednotlivých bloků mikroarchitektury,
- popis struktury mikroarchitektury.

Jazyk CodAL a nástroje, které ho zpracovávají, mají dále prostředky pro zajištění ekvivalence mezi vygenerovanými nástroji a hardwarem, pro zjištění kritických sekcí (přístup ke zdroji ze dvou funkčních jednotek v jednom cyklu) a pro zjednodušení verifikace výsledného hardwarového popisu [4].

2.2.1 Popis instrukce

Základními bloky modelu v jazyce CodAL jsou elementy (`element`) a události (`event`). Při návrhu mikroprocesoru lze využít elementy, kde každý element reprezentuje jednu instrukci (*instruction-set model*), nebo využít rozložení modelu na události a popsat mikroprocesor na úrovni časování (*cycle accurate model*). Z hlediska této práce je důležitější popis na úrovni instrukcí.

Syntaxe elementu popisující instrukci (se dvěma operandy) je v následující ukázce:

```
element nazev_instrukce
{
        use reg as dst, src;

        assembler { ... };

        binary { ... };

        semantics
        {
          ...
        };
}
```

V kódu jsou vidět čtyři sekce uvozené klíčovými slovy `use`, `assembler`, `binary` a `semantics`. Jedná se popořadě o sekce lokální deklarace, sekce assembleru, binární sekce a sekce sémantiky.

Sekce lokální deklarace umožňuje instanciovat jiné elementy a skupiny u instrukcí například jako operandy,

sekce assembleru obsahuje specifikaci instrukce v textové formě, využívá se ke generování assembleru a disassembleru,

binární sekce obsahuje binární reprezentaci instrukce pro tvorbu strojového kódu,

sekce sémantiky obsahuje chování popsané v ANSI C s výjimkou ukazatelů, struktur, výčtů, příkazu skoku, příkaz `switch` nemůže obsahovat cykly a větvení (to je jinak povoleno).

2.2.2 Instrukční rozšíření

Procesory s aplikačně specifickým rozšířením instrukční sady obvykle obsahují univerzální instrukční sadu, která je rozšířena o další speciální instrukce. Základní instrukční sada může být z prostorových důvodů nebo kvůli příkonu zmenšena (často například odebráním FPU – jednotky pro práci s plovoucí řádovou čárkou).

Rozšiřující instrukce jsou navrženy tak, aby umožňovaly procesoru efektivní práci v dané oblasti, kde má být nasazen. Může se jednat o oblast zpracování signálů – audio, video, detekce obrazu, telekomunikace, software radio, bezdrátové technologie a pod.

V sekci sémantiky jazyka CodAL se popisuje chování instrukce v jazyce C. Díky tomuto jazyku je navrhování a implementace nových instrukcí jednoduché a rychlé, není třeba navrhovat obvod jako takový, ale algoritmem popsat řešení daného problému. Ukázka aplikačně specifické instrukce v jazyce CodAL je v příloze B.

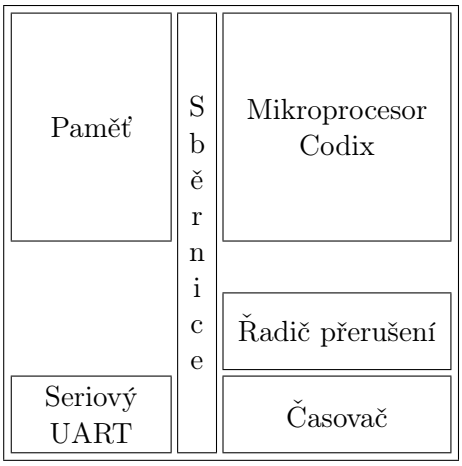
2.3 Codix

V rámci projektu Lissom[1] byl navržen SoC (obrázek 2.2) s mikroprocesorem Codix. Jedná se o 32bitový mikroprocesor inspirovaný architekturou ARM. Návrh kladl důraz na univerzalitu

[1] Jedná se o projekt, který běží na FIT VUT v Brně

a možnosti dalšího rozšíření.

Instrukční sada mikroprocesoru Codix je popsána v jazyce CodAL, ze kterého je následně vygenerován popis hardwaru. Právě popis v jazyce CodAL umožňuje rychle provádět změny, ve velké míře konfigurovat instrukční sadu Codixu a přizpůsobit ho dané oblasti nasazení [12].



Obrázek 2.2: Architektura SoC Codix

2.3.1 Architektura

Procesor je založen na architektuře s redukovanou instrukční sadou – architektura RISC. Není využíváno mikrokódu, operace jsou v procesoru realizovány obvodem. Zároveň byla snaha o co největší jednoduchost procesoru na obvodové úrovni. Aby byl hardware co nejjednodušší, jsou všechny problémy, u kterých je to možné, řešeny na úrovni softwaru při překladu.

Základní datový typ procesoru je 32bit integer. Primární programovací jazyk je C. Assembler není optimalizován pro programování člověkem, nicméně lze ho využít (to zejména v případě rozšíření procesoru o SIMD instrukce [12]).

Procesor je na úrovni programovacího jazyka C kompatibilní s architekturou ARM. Dále je procesor navržen tak, že program a data sdílejí společný paměťový prostor.

Kapitola 3

FPGA a dynamická rekonfigurace

Programovatelná hradlová pole (FPGA – Field Programmable Gate Array) jsou specializované integrované obvody, jejichž funkcionalita není dána již při výrobě. Návrh FPGA je proveden tak, aby se čip dal konfigurovat dle potřeby až samotným zákazníkem. Odlišují se tím od ASIC, u kterých je funkcionalita známá předem a dána napevno ve výrobě. Čipy FPGA poskytují největší flexibilitu z programovatelných logických zařízení (PLD – programmable logic device). Možnost změnit konfiguraci staví FPGA mezi univerzální procesory a ASIC.

Flexibilita FPGA je ale spojena s několika nevýhodami ve srovnání se zákaznickými obvody. V ASIC je integrovaný obvod vyhotoven přímo na úrovni tranzistorů. V FPGA jsou na úrovni tranzistorů vyhotoveny konfigurovatelné bloky a obvod, jehož funkcionalitu FPGA zastává, je výsledkem nastavení těchto konfigurovatelných bloků.

Tato „mezivrstva“ navíc u FPGA přináší několik nevýhod ve srovnání s ASIC. Plocha potřebná pro danou funkcionalitu je u FPGA obvykle 20krát až 35krát větší. Klesá výpočetní výkon, který je 3krát až 4krát nižší. S větším počtem potřebných tranzistorů roste navíc i spotřeba elektrické energie. Ta je 10krát vyšší než u zákaznických obvodů stejné funkcionality [7].

Proti těmto nevýhodám stojí cena a doba návrhu FPGA čipu. Tam, kde návrh ASIC stojí miliony a trvá měsíce až roky, lze FPGA pořídit o tři až čtyři řády levněji a obvod navrhnout v horizontu několika dnů. Proto je FPGA hojně vyžíváno jako alternativa k ASIC i v případě, že se neplánuje využívat rekonfigurace [3].

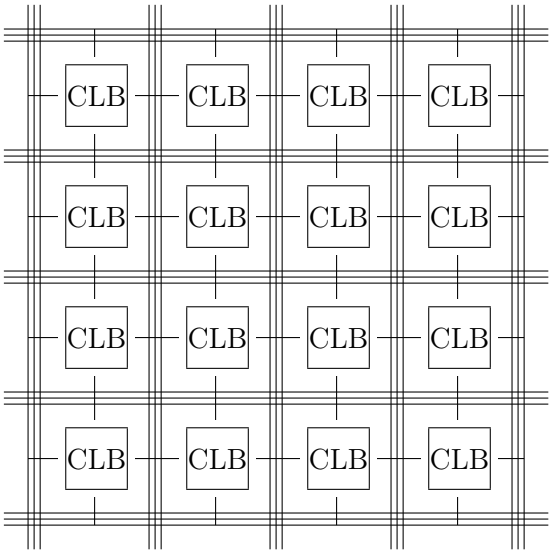
3.1 Struktura a programování FPGA

Vnitřní struktura FPGA je složena z 2D matice konfigurovatelných logických bloků (CLB - configurable logic block), obvodů řízení hodinového signálu, blokové RAM, multiplikátorů a dalších prvků [8]. Schéma vnitřní struktury FPGA je znázorněno na obrázku 3.1.

Každý konfigurovatelný logický blok je složen z[1]:

- carry řetězce pro tvorbu rychlých aritmetických jednotek,
- rychlého propojení k sousedním CLB,
- několika slice bloků.

[1]Složení jednotlivých CLB se liší výrobce od výrobce. Tato práce je založena na produktech společnosti Xilinx a popis vnitřní struktury je vztažen právě k těmto produktům



Obrázek 3.1: Struktura FPGA

Slice je elementární programovatelný logický blok, který se dále dělí na:

- funkční generátory,
- registry,
- pomocnou logika (aritmetika, carry),
- multiplexory.

Počet funkčních generátorů se liší podle modelu FPGA, typicky dva a více (např. FPGA čip Virtex-5 má 4). Funkční generátory jsou realizovány vyhledávací tabulkou (LUT – lookup table), která má 3 a více vstupů. Tento prvek se značí jako N-LUT (kde N je počet vstupů) a lze jej využít jako RAM paměť a posuvný registr. Nejčastějším využitím je však realizace libovolné binární funkce s daným počtem vstupů.

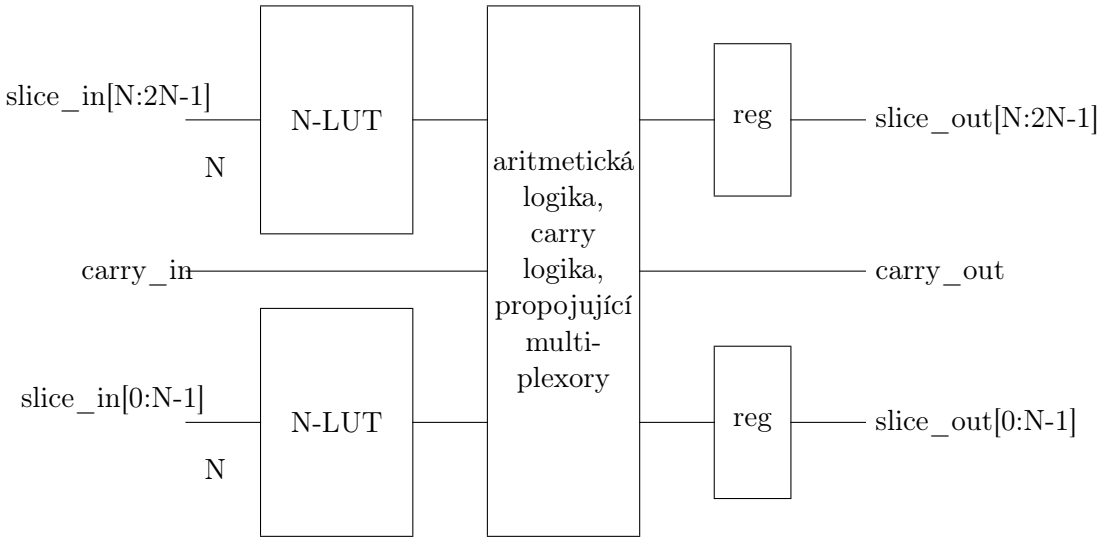
Kromě rychlého napojení k sousedním CLB mají tyto bloky také napojení na globální propojovací matici. Ta má na každém křížení vodičů propojovací strukturu zobrazenou na obrázku 3.3. Výsledné propojení je dáno nastavením programovatelných spínačů, což jsou tranzistory napojené na bitovou paměť, kde podle obsahu paměti je tranzistor otevřen nebo uzavřen. Na podobné paměti jsou napojeny i vnitřní prvky slice bloků. Konfigurace a tedy i výsledná funkcionalita FPGA je dána obsahem bitových pamětí. Tato bitová informace se nazývá *bitstream* a je vygenerována nástroji dodanými výrobcem čipu [3].

Generování výsledného bitstreamu je rozděleno na několik fází: logická syntéza, mapování, rozmístění, propojování a nakonec generování samotného bitstreamu[2].

Logická syntéza má jako vstup popis hardwaru v některém z HDL jazyků. Syntéza může proběhnout bez ohledu na výslednou architekturu použitého FPGA.

Mapování již zohledňuje výsledné FPGA, výstup ze syntézy je zde mapován na dostupné logické bloky cílového čipu.

[2]Proces zde ukázaný je typický [3], nicméně může se lišit podle výrobce



Obrázek 3.2: Schéma struktury slice bloku

Rozmístění je proces, při kterém jsou zvoleny konkrétní logické komponenty FPGA.

Propojování zajistí tvorbu potřebných spojení mezi konkrétními logickými bloky.

Generování bitstreamu je poslední fází. Výstup získaný z předchozích fází je zde převeden do formy bitové informace, která slouží k programování pamětí určujících konfiguraci logických bloků a ostatních prvků vnitřní struktury.

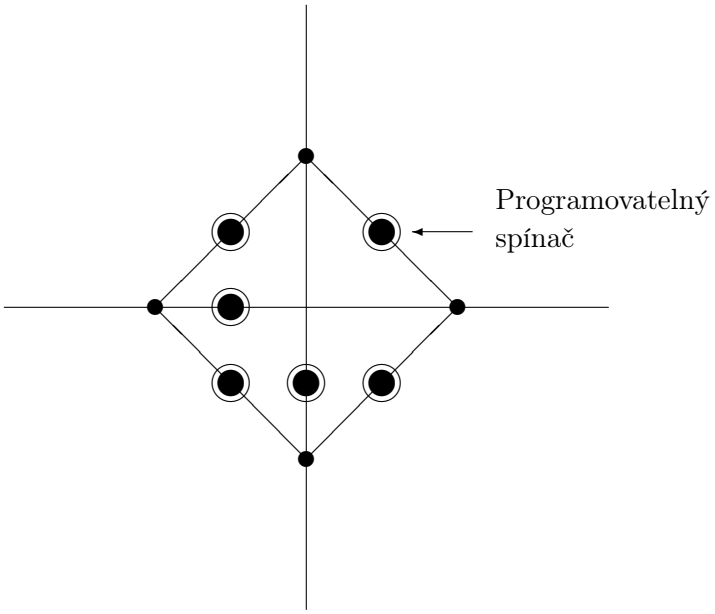
3.2 Dynamická rekonfigurace

Rekonfigurace je základní vlastností každého FPGA. Je při ní nahrán nový bitstream do paměti čipu, čímž dojde ke změně nastavení vnitřní struktury a celkové funkcionality. Statická rekonfigurace vyžaduje, aby bylo FPGA zastaveno a externí kontrolér mohl nahrát bitovou informaci. Zastavení FPGA znamená i přerušení výpočtu, který v tu dobu probíhal a tím pádem i pozdější získání výsledků.

Východisko z této situace poskytuje dynamická rekonfigurace. FPGA je prvotně nakonfigurováno plným bitstreamem a poté je možnost nahrát částečný bitstream, který změní konfiguraci jen určitého regionu na FPGA.

Čip se tak rozdělí na statickou část, která je celou dobu přerušení v provozu, a jednu nebo více dynamických částí, které se částečným bitstreamem rekonfigurují (obrázek 3.4).

Plocha rekonfigurovatelných oblastí není libovolná. Je omezená minimální adresovatelnou oblastí v konfigurační paměti zvanou rámec (frame). Velikost této oblasti přímo určuje minimální rekonfigurovatelnou plochu a liší se podle modelu FPGA čipu a jeho výrobce, např. čip Virtex-5 má rámce o velikosti 20 CLB na výšku a 1 CLB na šířku [14].

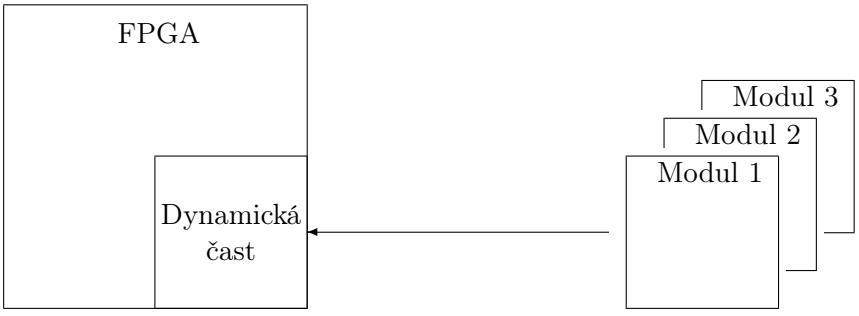


Obrázek 3.3: Způsob propojení vodičů na křížení

Díky možnostem měnit části obvodu za chodu, je možné zmenšit plochu potřebnou pro implementaci. Nutný základ je ve statické části a rozšiřující možnosti jsou ve formě dynamicky rekonfigurovatelných modulů. Menší plocha použitá pro implementaci daného obvodu znamená menší spotřebu a nižší cenu výsledného návrhu. Návrh je navíc univerzálnější, a to zejména z důvodu možnosti implementovat více funkctionality, která by se při statickém návrhu nevešla do prostorových (a tím pádem i cenových) omezení. Další výhodou návrhu využívající rekonfiguraci je tvorba systémů odolných proti selhání [14].

Dynamickou rekonfiguraci lze řídit dvěma způsoby:

- externě přes paralelní konfigurační přístupový port (PCAP),
- interně přes interní konfigurační přístupový bod (ICAP).



Obrázek 3.4: Schéma dynamické rekonfigurace

3.3 Dynamicky rekonfigurovatelný návrh

Návrh částečně dynamicky rekonfigurovatelného obvodu lze typicky rozdělit do několika kroků:

1. lokalizace dynamických částí,
2. nahrazení blackbox definicí,
3. syntéza statické části,
4. implementace chování dynamických částí,
5. syntéza dynamických částí.

Nahrazení blackbox definicí znamená definování rozhraní a vynechání popisu vnitřního chování. Statické a dynamické části jsou syntetizovány každá zvlášť a později spojeny – tzv. bottom-up syntéza.

3.4 Rozhraní statické a dynamické části

Statická a dynamická část FPGA musí být propojena přes neměnící se rozhraní. Bloky definované jako blackbox nejsou implementovány, zatímco ostatní jsou. Chybějící logika blackbox modulu má dopad na určování především časovacích omezení návrhu.

Řešením je implementovat minimální logiku, obvykle stačí vstupy a výstupy napojit na registry. Výhodou tohoto přístupu je přesnější určení časovacích omezení.

Druhou možností je vložení logiky na každé spojení – tzv. proxy logiky. Ta je obvykle implementována jako logický člen LUT1, časovací omezení není tak přesné jako v případě minimální logiky blackbox modulu, ale o vložení na potřebná místa se starají automatizované nástroje [13].

3.5 Dopad na výkon

Přidávání sekcí při využívání částečné dynamické rekonfigurace s sebou nese negativní dopad na výkon a plochu.

Vzhledem k přidané režii (jako je například výše zmíněná proxy logika) je doporučeno počítat s 10% snížením frekvence hodinového signálu. Co se plochy FPGA týče, je doporučeno neočekávat hustotu zaplnění vyšší jak 80%.

3.6 PCAP

Dynamická rekonfigurace řízená přes paralelní konfigurační přístupový bod (PCAP – parallel configuration access port) probíhá u zařízení Xilinx obvykle přes rozhraní JTAG, Serial Mode nebo SelectMap [14].

JTAG je název užívaný pro standardizované rozhraní IEEE 1149.1 Standard Test Access Port, používané pro programování zařízení a testování plošných elektrických obvodů,

Serial Mode je rekonfigurace přes sériové rozhraní SPI v módu slave,

SelectMap je proprietární rozhraní od společnosti Xilinx používané ke konfiguraci jejichž zařízení.

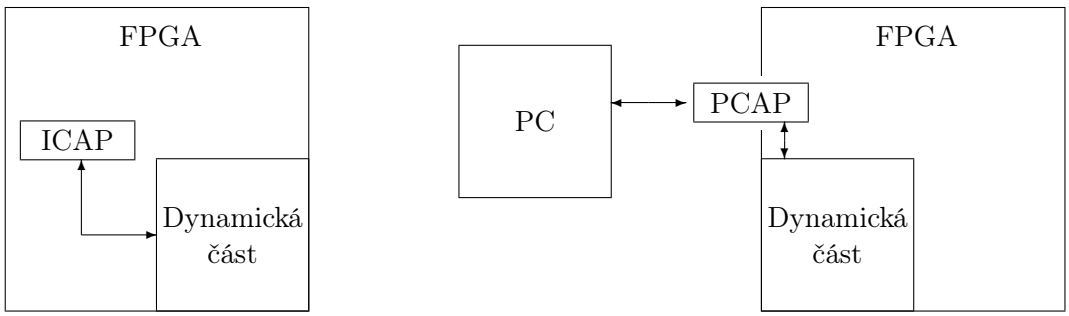
Rekonfigurované FPGA je propojeno s vnějším zařízením (typicky počítačem), které iniciuje spojení, řídí a kontroluje nahrávání částečného bitstreamu.

V případě použití JTAG stačí mít FPGA připojené přes programátor k počítači. U SelectMap je třeba definovat navíc omezení v UCF souboru a zohlednit SelectMap při generování bitstreamu.

3.7 ICAP

Další způsob řízení dynamické rekonfigurace je přes interní konfigurační přístupový bod (ICAP – Internal Configuration Access Port). Ve své podstatě se jedná o interní variantu řízení rekonfigurace přes PCAP.

Rekonfigurace řízená interně vyžaduje dodatečné zásahy do statické části FPGA. Ta musí být rozšířena o instanci ICAP rozhraní. Vzhledem k internímu řízení je třeba dále připojit jednotku, která bude rekonfiguraci řídit. Jedná se buď o stavový automat nebo o mikroprocesor jako je Microblaze či PowerPC 405 [14]. Srovnání PCAP a ICAP je znázorněno na obrázku 3.5.



Obrázek 3.5: Srovnání ICAP (nalevo) a PCAP (napravo)

Konfigurační mód	Max. frekvence	Šířka dat	Max. rychlost
ICAP	100 MHz	32 bit	3.2 Gb/s
SelectMap	100 MHz	32 bit	3.2 Gb/s
Serial Mode	100 MHz	32 bit	100 Mb/s
JTAG	66 MHz	1 bit	66 Mb/s

Tabulka 3.1: Rychlosti jednotlivých konfiguračních rozhraní

3.8 Rychlost (re)konfigurace

Každý konfigurační mód s sebou nese určitá omezení maximální frekvence a datové šíře při přenosu bitstreamu do zařízení. Obě tyto vlastnosti konfiguračního rozhraní určují maximální rychlost přenosu bitstreamu. Rychlosti jednotlivých rozhraní jsou v tabulce 3.1.

Kapitola 4

Návrh rekonfigurovatelné jednotky

Úkolem praktické části této bakalářské práce je navrhnout rekonfigurovatelnou jednotku (její strukturu a změnu za chodu), která bude rozšiřovat mikroprocesor Codix.

4.1 Cíl

Mikroprocesor Codix byl navržen jako ve velké míře upravitelný aplikačně specifický procesor. Díky využití jazyka CodAL je možné rychle reagovat na potřeby nového nasazení procesoru a navrhnout nové instrukční rozšíření.

Motivace rozšířit procesor Codix o rekonfigurovatelnou funkční jednotku je dvojí. V prvním případě je návrh tvořen *a priori* před nasazením v produkci, Zde však existuje riziko, že specifikace požadavků bude neúplná nebo nepřesná. Pokud by v takovém případě byl Codix vyhotoven „v křemíku“ jako ASIC nebyla by už jiná možnost jak změnit funkcionalitu procesoru než navrhnout novou revizi a vyrobit nové kusy V druhém případě, při použití Codixu v FPGA, kde by mohla proběhnout rekonfigurace v případě změn návrhu, je potřeba zohledňovat potřebnou plochu – větší plocha znamená vyšší cenu a vyšší spotřebu elektrické energie. Některé instrukce proto mohou mít formu modulů pro rekonfigurovatelnou jednotku a šetřit tak místo se zachováním potřebné univerzality.

4.2 Rozhraní a napojení na Codix

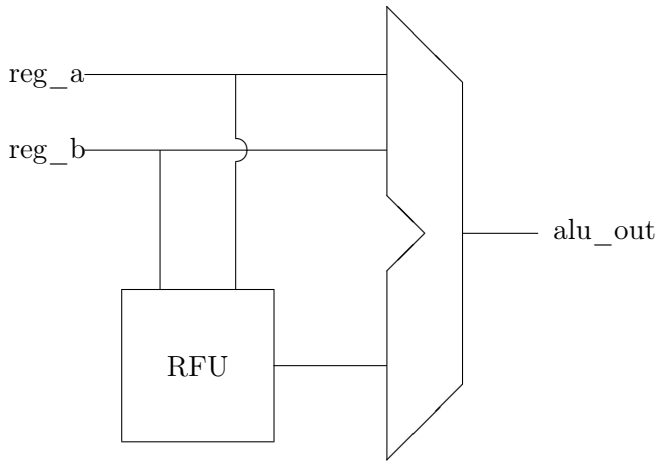
Možnosti napojení na Codix jsou v principu dvě:

- externí
- interní

První možnost, externí napojení rekonfigurovatelné jednotky, znamená vytvoření periferního zařízení umístěné do SoC mezi ostatní. Tato periferie by byla napojena přes systémovou sběrnici do jádra procesoru. Komunikace s externě připojenou jednotkou může probíhat dvěma způsoby. První je za pomoci dodatečně implementovaných instrukcí typu IN a OUT. Druhá možnost je vstupní signály rekonfigurovatelné jednotky namapovat do operační paměti.

Při napojení jednotky interně je potřeba jiného přístupu. Je třeba zvolit místo napojení a způsob komunikace. Jako vhodné místo se jeví předsazení (rozšíření) aritmeticko-logické jednotky (ALU – arithmetic logic unit).

ALU by byla rozšířena o další vstup, na který by byl připojen výstup z rekonfigurovatelné jednotky. Vstupní signály (operandy) by byly přivedeny jednak do ALU a jednak do



Obrázek 4.1: Schéma zapojení rekonfigurovatelné funkční do ALU procesoru Codix

rekonfigurovatelné jednotky. Dále by byla vytvořena nová aritmetická nebo logická instrukce, která by v ALU přepisovala vstup z rekonfigurovatelné jednotky na výstup.

Pro tuto práci byl zvolen druhý přístup – napojení jednotky přímo dovnitř jádra procesoru. Toto zapojení vyžaduje minimum zásahů do procesoru a zároveň zachovává transparentnost vykonávaných operací v rekonfigurovatelné jednotce. Způsob tohoto zapojení je znázorněn na obrázku 4.1.

Rozhraní rekonfigurovatelné jednotky tvoří dva vstupy a jeden výstup. Minimální počet připojených signálů je vhodný, neboť se tak zajistí co nejmenší ovlivnění funkčnosti procesoru při rekonfiguraci. Během doby rekonfigurace nebude samozřejmě možné používat rozšiřující jednotku, ale zbytek procesoru bude v provozu a k dispozici.

4.3 Časování

Procesor Codix využívá zřetězeného principu zpracování – pipelining. Linka je v procesoru rozdělena na části získání instrukce z paměti (FE – fetch), dekódování instrukce (DE – decode), provedení samotné operace (EX – execute) a nakonec zápis výsledku zpět do paměti (WB – writeback).

V procesoru jsou obsaženy pipeline registry. Ty jsou napojeny na hodinový signál a signál řídícího obvodu, který povoluje činnost jednotlivých registrů. Registry oddělují jednotlivé sekce na lince a je přes ně synchronizována činnost procesoru. Mezi registry je obsažena kombinační logika, ke které není přiveden hodinový signál a je na něm nezávislá.

ALU procesoru, a jí předřazená rekonfigurovatelná jednotka, se nachází v části EX – execute. Z obrázku 4.1 je zřejmé, že doba propagace signálu bude různá v závislosti na použití rekonfigurovatelné jednotky. Operace provedená čistě na ALU (aritmetické a logické operace) bude vyžadovat méně času než operace provedená v rekonfigurovatelné jednotce.

V jazyce CodAL lze definovat dobu trvání jednotlivých operací a samotný procesor Codix je navržen tak, že hazardy nejsou řešeny na HW úrovni, ale řeší je překladač [12].

4.4 Způsob rekonfigurace

V kapitole věnující se FPGA a dynamické rekonfiguraci bylo popsáno, že rekonfigurace může být řízena interně nebo externě. Pro tuto práci bylo zvoleno externí řízení rekonfigurace přes JTAG – vývojová deska s FPGA je stále připojena k počítači, je možné mít rekonfiguraci neustále pod dohledem a zároveň tak zjednodušit ladění a hledání chyb.

Rekonfigurace přes JTAG je vyvolána požadavkem na naprogramování částečného bitstreamu a je ukončena, jakmile byl zaslán příkaz `DESYNCH` [14]. Pro rozpoznání konce rekonfigurace je proto třeba vhodným softwarem kontrolovat zaslané příkazy.

4.5 Rozšíření instrukční sady

Aby bylo možné rekonfigurovatelnou jednotku používat, je třeba rozšířit instrukční sadu procesoru. Vzhledem k vybranému napojení stačí rozšířit instrukční sadu o jednu instrukci – ta se bude transparentně chovat jako další aritmetická nebo logická operace prováděná v ALU. Z hlediska strojového kódu a assembleru se jedná o novou číselnou konstantu resp. symbolickou instrukci, z hlediska jazyka C lze na novou instrukci hledět jako na speciální operaci (v následujícím kódu viz funkci `foo()` a operaci „křížek“) nebo jako na funkci dvou proměnných (viz funkci `bar()` a funkci `rfu_op()`):

```
int foo(int a, int b){
        return a # b;
}

int bar(int a, int b){
        return rfu_op(a, b);
}
```

Možnost využívající speciální funkci by mohla být realizována jako makro preprocesoru jazyka C, které na dané místo vloží inline assembler provádějící danou operaci.

Kapitola 5

Implementace rekonfigurovatelné jednotky a modulů

Tato kapitola se zabývá praktickou implementací navržené rekonfigurovatelné jednotky ve vývojovém prostředí od společnosti Xilinx a jazyce VHDL.

5.1 Rozhraní jednotky

Rekonfigurovatelná část návrhu je ve VHDL definována jako blackbox komponenta – je specifikováno pouze rozhraní této části. V jazyce VHDL je rozhraní definováno jako entita.

Vstupem jednotky jsou dva 32bitové signály a výstupem signál o stejné šíři. Ty jsou implementovány jako vstupní resp. výstupní `std_logic_vector()`.

5.2 Napojení na ALU

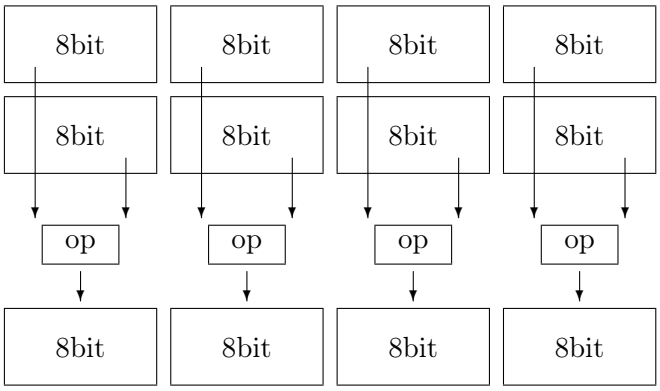
Aritmeticko-logická jednotka mikroprocesoru Codix má předem vypočítané výsledky ve formě vnitřních signálů. Vstupní signál nesoucí operaci je přiveden na multiplexor, který vybere odpovídající z vnitřních signálů a na výstup předá výsledek dané operace.

Vstupní sada signálů ALU je rozšířena o jeden další. Ten obsahuje výsledek operace provedené v rekonfigurovatelné jednotce a je přiveden k multiplexoru volícímu výsledek.

Rekonfigurovatelná jednotka je implementována kombinační logikou a má na svém výstupu neustále k dispozici předpočítaný výsledek. Ten pak stačí podle signálu s operací propagovat přes multiplexor na výstup ALU.

5.3 Architektura modulů

Moduly implementují chování a využívají rozhraní dané entitou rekonfigurovatelné jednotky. Pro tuto práci byla zvolena implementace modulů provádějící operaci typu SIMD – single instruction multiple data. Vstupní 32bitové signály (slova) jsou každý rozdělen na čtyři 8bitové části (slabiky). Nad každým párem slabik je provedena operace a výsledek (4 slabiky) se zapíše do výstupního signálu. Ukázka SIMD je na obrázku 5.1.



Obrázek 5.1: Ukázka rozdělení 32bitového slova na 8bitové slabiky

5.4 Generování bitstreamu a rekonfigurace

Statická část a dynamické moduly jsou syntetizovány zvlášť v programu Xilinx ISE. Floor planning (mapování, rozmístění a propojování) společně s volbou velikosti plochy vyhrazené pro moduly je proveden v nástroji PlanAhead. Výsledkem je bitstream pro konfiguraci statické části a částečné bitstreamy pro každý modul.

Vývojová karta je připojena přes JTAG rozhraní k počítači a konfigurace statické části a následná rekonfigurace dynamické části je řízena softwarem iMPACT.

Kapitola 6

Výsledky z použití rekonfigurovatelné jednotky

V této kapitole je popsáno použití rekonfigurovatelné jednotky, testovací program a testování. Nakonec jsou shrnuty výsledky rekonfigurace za běhu programu.

6.1 Testovací programy a průběh testování

Pro otestování rekonfigurace jednotky za běhu byl zvolen program, který cyklicky rozsvěcuje jednu z osmi LED diod připojených k procesoru Codix. Zdrojový kód programu je v příloze C.

Správné provedení částečné dynamické rekonfigurace je provedeno tak, že do FPGA je naprogramován procesor Codix rozšířený o rekonfigurovatelnou jednotku. Do procesoru je nahrán program pro práci s LED diodami a následně je vyvolána rekonfigurace jednotky naprogramováním jednoho modulů. V případě správné částečné rekonfigurace není průběh vykonávaného programu přerušen.

Otestování funkčnosti rekonfigurovatelné jednotky je provedeno programem pracujícím se SIMD rozšířením implementovaným ve dvou dynamických modulech. V modulu A je SIMD rozšíření provádějící nad každou slabikou ve slově operaci:

$$výsledek = slabikaA + slabikaB$$

V modulu B je operace sčítání nahrazena odčítáním:

$$výsledek = slabikaA - slabikaB$$

Program, demonstrující funkčnost rekonfigurovatelné jednotky, je napsán v jazyce C a přeložen překladačem vygenerovaným z popisu procesoru Codix v jazyce CodAL.

Statická část nemá chování rozšiřující jednotky definováno, a proto před nahráním programu do procesoru je nutno mít v rekonfigurovatelné jednotce jeden z dynamických modulů. Program je tedy nahrán do procesoru až po správném provedení rekonfigurace jednotky. Výsledek je poté binárně zobrazen rozsvícením LED diod na vývojové desce. Kód programu je následující:

```
int main()
{
  volatile unsigned int i, o = 0;
  unsigned reg_a = 0, reg_b = 0, reg_out;

  reg_a = reg_a | 1; reg_a = reg_a << 8;
  reg_a = reg_a | 1; reg_a = reg_a << 8;
  reg_a = reg_a | 1; reg_a = reg_a << 8;
  reg_a = reg_a | 1;

  reg_b = reg_b | 16; reg_b = reg_b << 8;
  reg_b = reg_b | 32; reg_b = reg_b << 8;
  reg_b = reg_b | 64; reg_b = reg_b << 8;
  reg_b = reg_b | 128;

  while (1)
  {

    __asm__ __volatile__ (
            "%0 = nor %1, %2"
            : "=r"(reg_out) : "r"(reg_a), "r"(reg_b)
    );

    for(o=0; o<=24; o+=8){
      __asm__ __volatile__(
              "print_reg %0"
              :: "r"((reg_out >> o) &0xFF));

      for (i=0; i<10000000; i++){
        __asm__ __volatile__("nop");
      }
    }

  }

  return (0);
}
```

Do 32bitové proměnné `reg_a` jsou za sebou uloženy čtyři 8bitové slabiky obsahující každá číslo 1. Do `reg_b` jsou uloženy slabiky obsahující popořadě čísla 16, 32, 64 a 128. Nad těmito proměnnými je provedena operace NOR, která byla vybrána, aby fungovala jako instrukce pracující s rekonfigurovatelnou jednotkou (překladač zatím nemá přímou podporu pro tuto operaci). Výsledek je poté postupně zobrazován na LED diodách jako binární číslo.

Očekávané výsledky jsou v tabulce 6.1. Modul B pracuje s operací odčítání, v ukázkovém programu se odčítá vetší číslo od menšího. Výsledky jsou záporné, přetečou a zobrazí se jako kladná čísla.

	Modul A	Modul B
Slabika 1	129	129
Slabika 2	65	193
Slabika 3	33	225
Slabika 4	17	241

Tabulka 6.1: Očekávané výsledky z použití dynamických modulů

6.2 Rychlost rekonfigurace

Doba trvání rekonfigurace byla spočítána ze vztahu:

$$čas = \frac{velikost\ v\ bitech}{rychlost\ rozhraní}$$

V tabulce 6.2 jsou naměřené a vypočtené doby trvání rekonfigurace. Měřeno bylo v nástroji iMPACT, který vypisuje dobu jakou trvalo programování čipu. Bitstream měřeného modulu má velikost jen 24 450 bytů a trvání rekonfigurace je zřejmě tak krátké, že je mimo rozlišovací schopnosti nástroje iMPACT.

	čas
Teoretický čas	0.37 ms
Naměřený čas	0 s

Tabulka 6.2: Teoretické a naměřené doby trvání rekonfigurace

6.3 Výsledky z použití a testování

Do FPGA byla naprogramována statická část procesoru Codix. Poté byl nahrán a spuštěn program na rozsvěcování LED diod. Za běhu procesoru bez přerušení programu byl do rekonfigurovatelné funkční jednotky nahrán modul A.

Program pro rozsvěcování LED diod běží v nekonečné smyčce a pro funkčnost bootloaderu, který běží na samotném procesoru, bylo třeba procesor resetovat. Poté byl nahrán program pro otestování modulů. Výsledky zobrazené na diodách při použití modulu A jsou v tabulce 6.3.

	binárně	decimálně
Slabika 1	10000001	129
Slabika 2	01000001	65
Slabika 3	00100001	33
Slabika 4	00010001	17

Tabulka 6.3: Výsledky z použití modulu A

Následně byl do funkční jednotky nahrán modul B. Během rekonfigurace jednotky byl procesor v provozu a program zobrazoval výsledky na LED diodách. Po doběhnutí zobrazovacího cyklu byly v rekonfigurovatelné jednotce vypočítány a poté zobrazeny již nové výsledky. Výsledky z použití modulu B jsou v tabulce 6.4.

	binárně	decimálně
Slabika 1	10000001	129
Slabika 2	11000001	193
Slabika 3	11100001	225
Slabika 4	11110001	241

Tabulka 6.4: Výsledky z použití modulu B

Naměřené hodnoty v tabulkách odpovídají očekávaným výsledkům z tabulky 6.1. Funkčnost jednotky a dynamických modulů je tím otestována.

Procesor byl před implementací rekonfigurovatelné jednotky taktován na 100 MHz. Po implementaci jednoty byla nástrojem PlanAhead stanovena maximální frekvence na 100.271 MHz (respektive minimální perioda 9.973 ns). Procesor tedy nebyl dodatečnými logickými prvky zpomalen.

Kapitola 7

Možnosti dalšího rozvoje

Použití řízení rekonfigurace přes JTAG je diskutabilní. Je sice sjednoceno programování procesoru Codix a rekonfigurace jednotky a je zde možnost kontrolovat průběh a výsledek rekonfigurace přes počítač, ale rozhraní JTAG je nejpomalejší z dostupných (viz tabulku 3.1). Rychlost nemusí být problémem u malých modulů, tak jak bylo prezentováno v předchozí kapitole, ale může se stát limitujícím faktorem při použití rozsáhlých modulů. Dále začne být rozhraní JTAG problémem, jakmile bude procesor Codix programován z paměti přímo na vývojové kartě (např. SysACE).

Náhradou za JTAG může být SelectMAP nebo ICAP. Obě rozhraní disponují mnohem větší přenosovou rychlostí. Interní řízení rekonfigurace přes ICAP by navíc poskytovalo plnou autonomii procesoru.

Zdá se, že ICAP je vhodným kandidátem při hledání náhrady za JTAG. Tento způsob řízení rekonfigurace ovšem vyžaduje změny v hardwaru – je třeba jednotky, která bude rekonfiguraci řídit. Možnosti jsou buď procesor (Microblaze, PowerPC) nebo stavový automat. Rozšiřovat procesor o procesor není vhodným způsobem, jak zařídit interní rekonfiguraci. Pokud tedy bude zvolen ICAP, bude třeba procesor Codix rozšířit o stavový automat. Dále je nutno navrhnout způsob, jak se bude rekonfigurace vyvolávat. Jako vhodná se jeví instrukce použitelná přímo v kódu programu. Nakonec je třeba zvolit paměť, odkud se budou částečné bitstreamy načítat.

Při použití Codixu na FPGA není s dynamickou rekonfigurací problém. V případě plánovaného vyhotovení Codixu jako zákaznický obvod, je třeba mít na paměti, že zachování rekonfigurovatelné jednotky vyžaduje mít k dispozici rekonfigurovatelný obvod. Při přístupu, kde je FPGA připojeno jako periferie k procesoru (podobně jako např. SoC Zynq od společnosti Xilinx), by byly třeba úpravy v návrhu a implementaci rekonfigurovatelné jednotky.

Nakonec je třeba zohlednit fakt, že hardwarový popis procesoru je generován automaticky z jazyka CodAL, zatímco implementace rekonfigurovatelné jednotky probíhala manuálně ve vygenerovaném kódu. V případě, že by nová verze Codixu (a tedy i nový vygenerovaný popis hardwaru) obsahovala změny v ALU, bylo by pravděpodobně nutno manuálně upravit kód rekonfigurovatelné jednotky. To poněkud ruší výhody automatického generování hardwarového popisu. Řešením by mohlo být například rozšíření jazyka CodAL o možnost definování rekonfigurovatelné jednotky jako black-box struktury, využít bottom-up přístupu a moduly specifikovat opět v jazyce CodAL.

Kapitola 8

Závěr

V této práci byl popsán možný přístup při rozšiřování aplikačně specifických procesorů. Konkrétně se jednalo o rozšíření mikroprocesoru Codix o rekonfigurovatelnou jednotku.

Byly shrnuty výhody souběžného návrhu hardwaru a softwaru, popsány vlastnosti jazyků pro popis architektury, ADL jazyk CodAL a dále byl popsán procesor Codix a jeho architektura.

Práce popsala možnosti dynamické rekonfigurace a její použití v oblasti souběžného návrhu hardwaru a softwaru, byly prozkoumány možnosti návrhu rekonfigurovatelné jednotky a bylo navrženo její rozhraní, napojení na procesor a způsob rekonfigurace.

Rekonfigurovatelná jednotka byla implementována, včetně dvou dynamických modulů se SIMD operací a testovacího programu. Změna funkce za běhu programu, funkčnost rekonfigurovatelné jednotky a dynamických modulů byla otestována spuštěním na vývojové kartě s FPGA čipem.

Nakonec byl diskutován možný rozvoj do budoucna, zejména z hlediska použitého rekonfiguračního rozhraní a související autonomie systému, vyhotovení Codixu jako ASIC a možné rozšíření ADL jazyka CodAL.

Literatura

[1] A Framework for Hardware-Software Co-Design of Embedded Systems. `http://embedded.eecs.berkeley.edu/Research/hsc/`, [cit. 2013-04-22].

[2] ApS Brno s.r.o.: CodAL Manual, reference guide, version 4.3. 2013.

[3] Hauck, S.; DeHon, A.: *Reconfigurable computing: the theory and practice of FPGA-based computation*. Morgan Kaufmann, 2008, ISBN 978-0-12-370522-8.

[4] Hruška, T.: Metodologie a vývojové prostředí pro rychlý návrh aplikačních procesorů. Přednáška na konferenci STECH: Moderní elektronické součástky, 2011.

[5] Husár, A.; Dolíhal, L.; Přikryl, Z.; aj.: Accelerating Sequential Computations Using Manually Discovered Instruction Set Extensions, 2013, Zatím nepublikovaný článek zaslaný v osobní korespondenci.

[6] Israelson, A.: TRANSCRIPT – Bill Gates and Steve Jobs at D5. `http://allthingsd.com/20070531/d5-gates-jobs-transcript/`, 2007-05-31 [cit. 2013-04-22].

[7] Kuon, I.; Tessier, R.; Rose, J.: Fpga architecture: Survey and challenges. *Foundations and Trends® in Electronic Design Automation*, ročník 2, č. 2, 2008: s. 135–253.

[8] Martínek, T.: Technologie FPGA. Přednáška z Návrhu číslicových systémů, 2011.

[9] Masařík, K.: *Systém pro souběžný návrh technického a programového vybavení počítačů: disertační práce*. Fakulta informačních technologií, Vysoké učení technické v Brně, 2008, ISBN 978-80-214-3863-7.

[10] Rivera, J.; van der Meulen, R.: Gartner Says Worldwide PC, Tablet and Mobile Phone Combined Shipments to Reach 2.4 Billion Units in 2013. `http://www.gartner.com/newsroom/id/2408515`, 2013-04-04 [cit. 2013-04-22].

[11] Teich, J.: Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, ročník 100, č. 13, 2012: s. 1411–1430.

[12] Výzkumná skupina Lissom na FIT VUT v Brně: Manuál procesoru Codix. 2012.

[13] Xilinx, Inc.: Hierarchical Design Methodology Guide. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_4/Hierarchical_Design_Methodology_Guide.pdf`, 2012 [cit. 2013-04-24].

[14] Xilinx, Inc.: Partial Reconfiguration User Guide. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_5/ug702.pdf`, 2012 [cit. 2013-04-24].

Příloha A

Obsah CD

./src zdrojové soubory dynamických modulů a testovacích programů

./bit plné a částečné bitstreamy (procesor Codix s rekonfigurovatelnou jednotkou a dynamické moduly)

./tex zdrojové soubory technické zprávy

./projekt.pdf technická zpráva

Příloha B

Ukázka aplikačně specifické instrukce

```
1  element i1_adpcm_coder
2  {
3          use REG as dst4, sd1, sd2, sd3;
4
5          assembler { dst4 "=" "i1_adpcm_coder" sd1 "," sd2 "," sd3
               "," SRC4_NAME "," SRC5_NAME };
6
7          binary { OPC9_ISE1:9 0:3 dst4 sd1 sd2 sd3 };
8
9          semantics
10         {
11                 int index, bufferstep, outputbuffer, outp_val,
                       indexTable_delta, delta;
12                 codasip_compiler_unused();
13                 index = regs[sd1];
14                 bufferstep = regs[sd2];
15                 outputbuffer = regs[sd3];
16                 indexTable_delta = regs[SRC4_INDEX];
17                 delta = regs[SRC5_INDEX];
18                 index += indexTable_delta;
19                 if ( index < 0 ) index = 0;
20                 if ( index > 88 ) index = 88;
21                 if ( bufferstep ) {
22                         outputbuffer = (delta << 4) & 0xf0;
23                 } else {
24                         outp_val = (delta & 0x0f) | outputbuffer;
25                 }
26                 bufferstep = !bufferstep;
27                 regs[sd1] = index;
28                 regs[sd2] = bufferstep;
29                 regs[sd3] = outputbuffer;
30                 regs[dst4] = outp_val;
31         };
32 }
```

Příloha C

Program na rozsvěcování LED

Autorem je Marián Pristach z FEKT VUT v Brně.

```
1  int main()
2  {
3          unsigned int x = 0x80;
4          volatile unsigned int i;
5
6          while (1) {
7
8                  __asm__ __volatile__("print_reg␣%0"::"r"(x));
9
10                 x = x >> 1;
11                 if (x == 0)
12                         x = 0x80;
13
14                 // wait
15                 for (i=0; i<1000000; i++) {
16                         __asm__ __volatile__("nop");
17
18                 };
19         }
20
21         return (0);
22 }
```